Estruturas de Dados

Árvore Vermelho e Preto

Universidade Estadual Vale do Acaraú – UVA

Paulo Regis Menezes Sousa

paulo_regis@uvanet.br

Árvores Vermelho e Preto

- As árvores Vermelho e Preto são árvores binárias de busca “aproximadamente” balanceadas.
- Também conhecidas como rubro-negras ou *red-black trees*.
- Foram inventadas por Bayer sob o nome “Árvores Binárias Simétricas” em 1972, 10 anos depois das árvores AVL.
- As árvores Vermelho e Preto possuem um campo extra para armazenar a cor de cada nó, que pode ser Vermelho ou **Preto**.

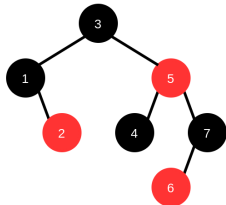
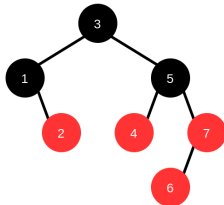
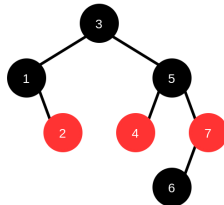
```
1  struct RB {
2      void *value;
3      RB *left;
4      RB *right;
5      RB *parent;
6      int color;
7  }
```

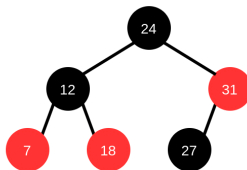
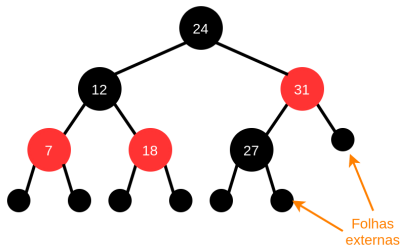
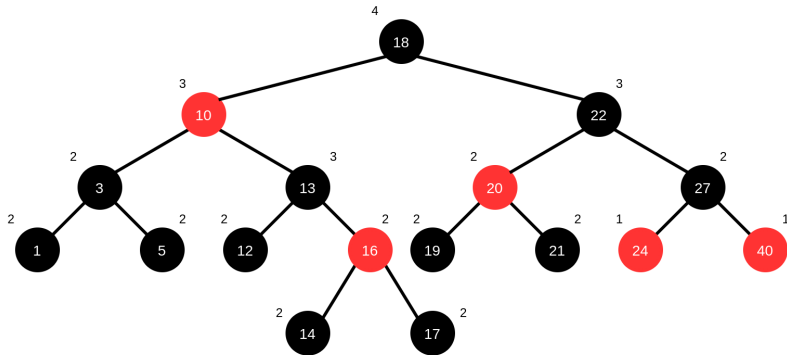
- Restringindo o modo como os nós são coloridos desde a raiz até uma folha, assegura-se que **nenhum caminho será maior que duas vezes o comprimento de qualquer outro**, dessa forma, a árvore é aproximadamente balanceada.
- Uma árvore Vermelho e Preto com n nós tem altura máxima

$$2\log(n+1)$$

- Por serem "balanceadas" as árvores Vermelho e Preto possuem complexidade logarítmica em suas operações

$$O(\log n)$$

- Uma árvore Vermelho e Preto é uma árvore de busca binária que satisfaz as seguintes condições:
  1. Todo nó é vermelho ou preto.
  2. A raiz é preta.
  3. Toda folha externa (`NULL`) é preta.
  4. Se um nó é vermelho, então ambos seus filhos são pretos
  5. Todos os caminhos a partir da raiz da árvore até suas folhas passa pelo mesmo número de nós pretos.

- Um nó que satisfaz as propriedades anteriores é denominado **equilibrado**, caso contrário é dito desequilibrado.
- Em uma árvore Vermelho e Preto todos os nós estão equilibrados.
- Uma condição óbvia obtida das propriedades é que num caminho da raiz até uma sub-árvore vazia **não pode existir dois nós vermelhos consecutivos**.

- Formas de Representação

- **Altura negra**: é número de nós negros encontrados até qualquer nó folha externo

Lema 1

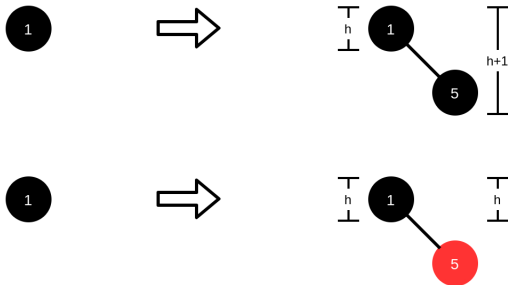
Seja x a raiz de uma (sub)árvore Vermelho e Preto, então ela terá no mínimo $2^{an(x)} - 1$ nós internos, onde $an(x)$ é a altura negra de x.

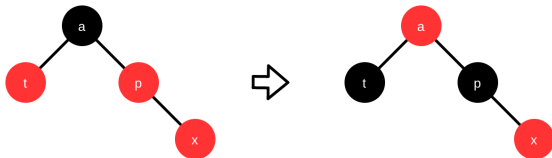
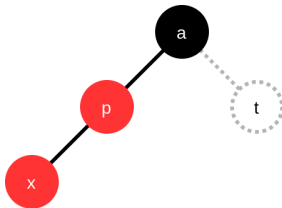
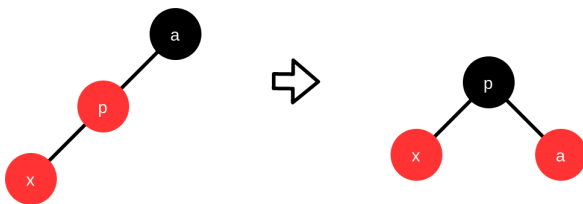
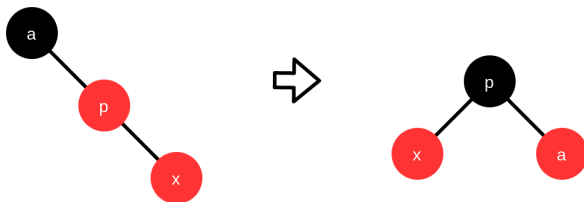
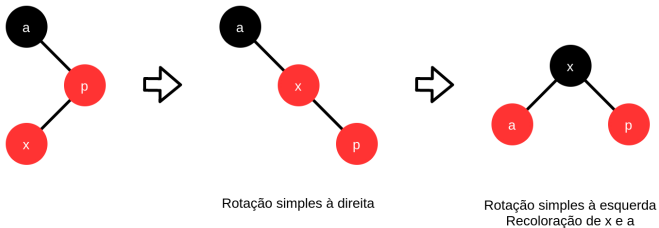
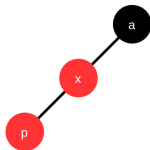
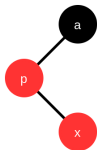
Lema 2

Uma árvore Vermelho e Preto com n nós tem no máximo altura $2\log_2(n+1)$

Inserção

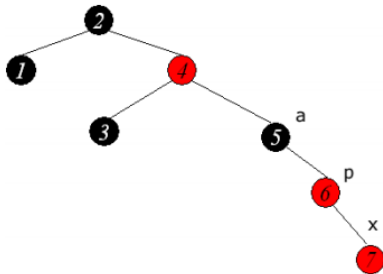
- Segue os princípios de uma inserção em Árvore Binária de Busca.
- Após a inserção um conjunto de propriedades é testado, e se a árvore não satisfizer essas propriedades, são realizadas **rotações** e/ou **ajustes de cores**, de forma que a árvore permaneça balanceada.

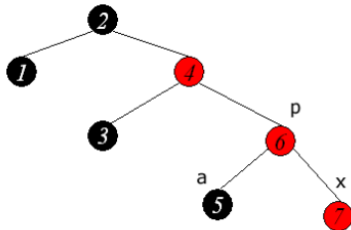
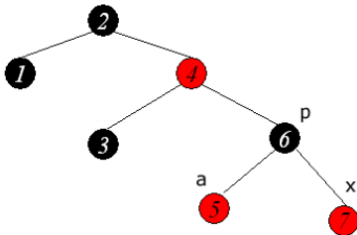
- Um nó é inserido sempre na cor vermelha, assim, não altera a altura negra da árvore.
- Se o nó fosse inserido na cor preta, invalidaria a propriedade (**5**), pois haveria um nó preto a mais em um dos caminhos

- Caso a inserção seja feita em uma árvore vazia, basta alterar a cor do nó para preto, satisfazendo assim a propriedade número (2).

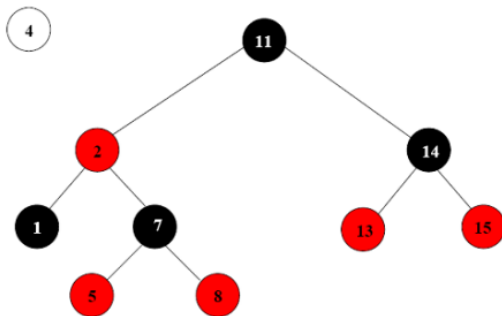
- Se o nó pai p é vermelho e o nó avô a é preto. Se t, o irmão de p (tio de x) é vermelho, ainda é possível manter o critério (**4**) apenas fazendo as recolorações de a, t e p.
- Se o pai de a é vermelho, o rebalanceamento tem que ser feito novamente considerando a como nó inserido.

- **Caso 2**: Suponha que p é vermelho, seu pai a é preto e seu irmão t é preto. Neste caso, para manter o critério (**4**) é preciso fazer rotações envolvendo a, t, p e x.
- Há 4 subcasos que correspondem às 4 rotações possíveis

- **Caso 2a**: O nó x é filho esquerdo de p, e p é filho esquerdo de a.
    - Aplicar Rotação Direita

- **Caso 2b**: O nó x é filho direito de p, e p é filho direito de a.
  - Aplicar Rotação Esquerda

- **Caso 2c**: O nó x é filho esquerdo de p, e p é filho direito de a
  - Aplicar Rotação Dupla Esquerda

- **Caso 2d**: O nó x é filho direito de p, e p é filho esquerdo de a
  - Aplicar Rotação Dupla Direita

Exemplo 1

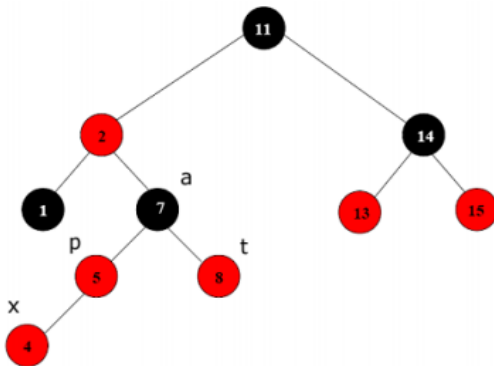
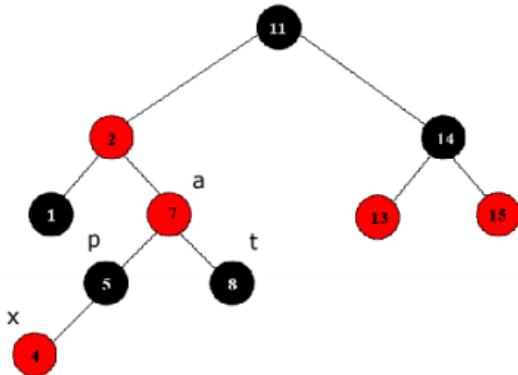
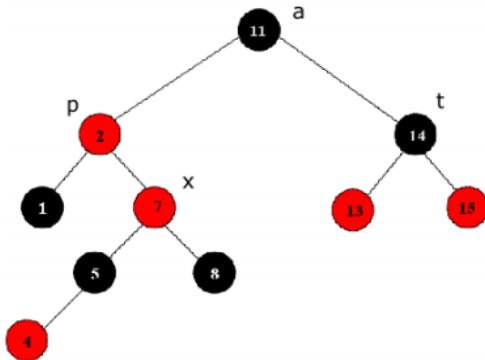
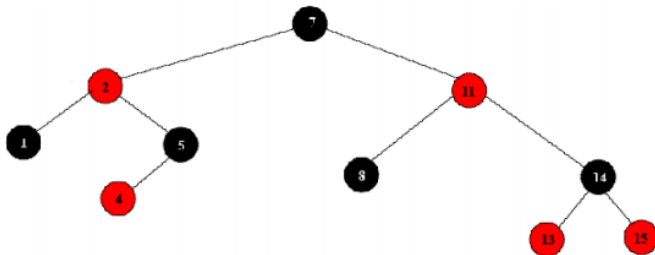
- Estado inicial da árvore
- Inserção do nó 7

- O tio t do elemento inserido x é preto, seu pai p é filho direito de a e x é filho direito de p.
  - **Caso 2b**: requer rotação esquerda em a

Exemplo 1

- Violação da propriedade pelos nós p e x – recoloração

Exemplo 1

- Recoloração dos nós p e x

- Estado inicial da árvore
- Inserção do nó 4

Exemplo 2

- O tio t do elemento inserido x é vermelho
  - **Caso 1**: requer a recoloração dos nós a, t e p
- Violação da propriedade (**4**)

Exemplo 2

- Nós p e t passam a ser pretos e o nó a passa a ser vermelho
- Violação da propriedade (**4**) entre os nós a e seu pai

- O tio t do elemento inserido x é preto e o elemento inserido é um filho da direita de p
  - **Caso 2d**: requer rotação dupla direita – rotação esquerda em p e rotação direita em x

- Processo termina porque já atingiu a raiz da árvore

```
void insercao_caso1(RB *x) {
    if (x->pai == NULL)
        x->cor = PRETA;
    else
        insercao_caso2(x);
}
```

```
void insercao_caso2(RB *x) {
    if (x->pai->cor == PRETA)
        ; /* Árvore ainda é válida */
    else
        insercao_caso3(x);
}
```

Inserção


```
1  void insercao_caso3(RB *x) {
2      RB *t = tio(x), *a;
3
4      if ((t != NULL) && (t->cor == VERMELHA)) {
5          x->pai->cor = PRETA;
6          t->cor = PRETA;
7          a = avo(x);
8          a->cor = VERMELHA;
9          insercao_caso1(a);
10     }
11     else {
12         insercao_caso4(x);
13     }
14 }
```

```
1  void insercao_caso4(RB *x) {
2      RB *a = avo(x);
3
4      if ((x == x->pai->direita) && (x->pai == a->esquerda)) {
5          rotacionar_esquerda(x->pai);
6          x = x->esquerda;
7      }
8      else if ((x == x->pai->esquerda) && (x->pai == a->direita)) {
9          rotacionar_direita(x->pai);
10         x = x->direita;
11     }
12     insercao_caso5(x);
13 }
```

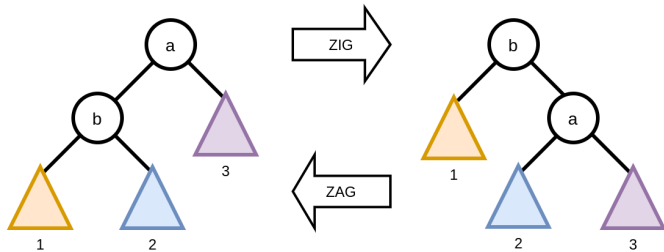
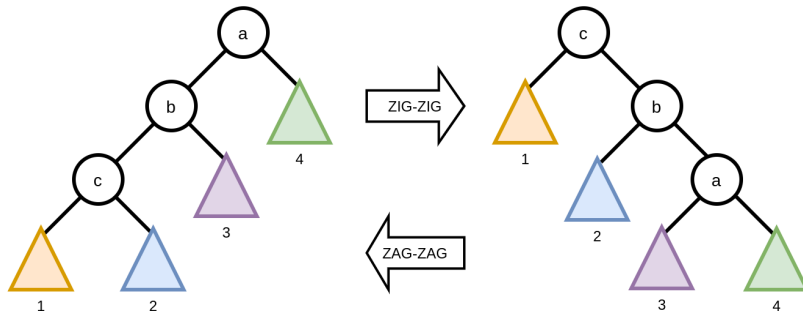
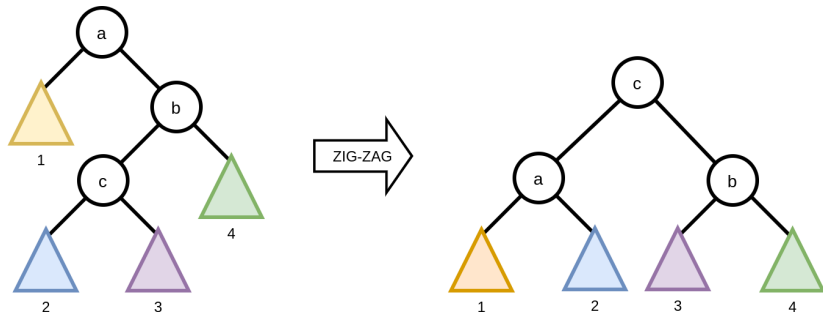
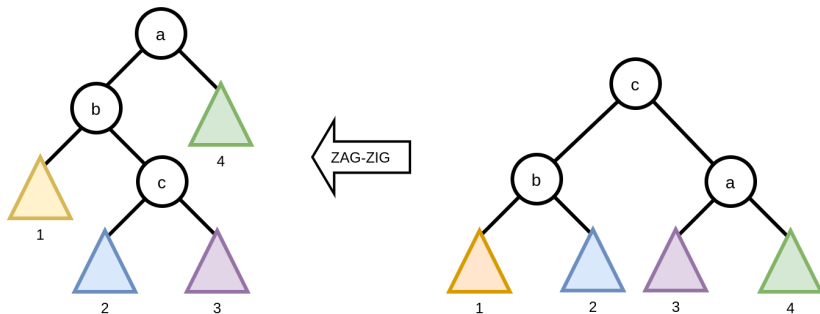
```c
void insercao_caso5(RB *x) {
    RB *a = avo(x);

    x->pai->cor = PRETA;
    a->cor = VERMELHA;

    if ((x == x->pai->esquerda) && (x->pai == a->esquerda)) {
        rotacionar_direita(a);
    }
    else if ((x == x->pai->direita) && (x->pai == a->direita)) {
        rotacionar_esquerda(a);
    }
}
```

Complexidade da Inserção

- Rebalanceamento tem custo $O(1)$
- Rotações têm custo $O(1)$
- Inserção tem custo $O(\log n)$

- A árvore splay foi inventada por **Daniel Sleator e Robert Tarjan em 1985**.
- Uma árvore splay é uma **árvore binária de busca autoajustável**.
- Árvores splay **mantêm equilíbrio sem qualquer condição explícita de equilíbrio**, como cor ou fator de balanceamento.
- Operações de **rotação** são executadas dentro da árvore **toda vez que um acesso é executado**.
  - aplicam-se rotações únicas em pares, cuja ordem depende dos vínculos entre filho, pai e avô.

- O custo *amortizado* de cada operação em uma árvore de n nós é $O(\log n)$.
- Quando um nó n é acessado, uma operação de splay é executada em n para movê-lo para a raiz.
  - Para executar uma operação de splay, realizamos uma sequência de rotações, que move n mais próximo da raiz.

O custo amortizado não deve ser confundido com custo médio. O conceito de **custo amortizado** se aplica a uma *sequência* de execuções de uma operação em que o custo de cada execução depende das execuções anteriores. Já o **custo médio** é calculado sobre um conjunto de execuções *independentes*.

Se pai(B) é raiz fazemos apenas uma rotação para esquerda ou direita.

Se pai(C) não é raiz e C e pai(C) são filhos do mesmo lado, fazemos duas rotações para direita ou duas rotações para a esquerda, no mesmo sentido começando pelo pai(pai(C)).

Se pai(C) não é raiz e C e pai(C) são filhos do lado oposto, faz uma rotação em pai(C) para direita e outra rotação no avô para esquerda de C.

Se pai(C) não é raiz e C e pai(C) são filhos do lado oposto, faz uma rotação em pai(C) para esquerda e outra rotação no avô para direita de C.